Symfony 5 : formulaires

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en Programmation par contrainte (IA)
Ingénieur en Génie logiciel

elmouelhi.achref@gmail.com

Plan

Symfony

Principe

- Les formulaires avec **Symfony** sont relatifs à des objets
- On peut définir un formulaire soit
  - dans le contrôleur
  - dans un autre objet qui sera appelé par le contrôleur

Pour commencer, nous considérons l'entité `Personne.php` avec les attributs suivants

```php
/**
 * @ORM\Entity(repositoryClass="App\Repository\PersonneRepository")
 */
class Personne
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $nom;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $prenom;
```

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Vérifier les modifications avec la console `MySQL` ou `phpMyAdmin`

Symfony

Création d'un formulaire

1. On appelle la méthode `createFormBuilder`
2. On spécifie le nom d'objet qui va assurer le binding
3. On indique le type **HTML** pour chaque attribut de l'objet
4. On appelle la méthode `getForm()` pour générer le formulaire
5. On passe la méthode `createView()` du formulaire à la vue

Ajoutons l'action suivante dans `PersonneController.php`

```
/**
 * @Route("/personne/add", name="personne_add")
 */
function addForm()
{
    $personne = new Personne();
    $form = $this->createFormBuilder($personne)
       ->add('nom', TextType::class, array('required' => true))
       ->add('prenom', TextType::class)
       ->add('save', SubmitType::class, ['label' => 'Ajouter une
          personne'])
       ->getForm();

    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Ajoutons l'action suivante dans `PersonneController.php`

```
/**
 * @Route("/personne/add", name="personne_add")
 */
function addForm()
{
    $personne = new Personne();
    $form = $this->createFormBuilder($personne)
       ->add('nom', TextType::class, array('required' => true))
       ->add('prenom', TextType::class)
       ->add('save', SubmitType::class, ['label' => 'Ajouter une
          personne'])
       ->getForm();

    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Les `use` nécessaires

```
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
```

Symfony

Les types de champs avec **Symfony**

- Les types de champs en **HTML** et **Symfony** ne sont pas forcément les mêmes
- La liste complète de différents types **Symfony** : `https://symfony.com/doc/current/reference/forms/types.html`

Symfony

Les types de champs avec **Symfony**

- Les types de champs en **HTML** et **Symfony** ne sont pas forcément les mêmes
- La liste complète de différents types **Symfony** : `https://symfony.com/doc/current/reference/forms/types.html`

Le champ `CSRF` : Cross Site Request Forgeries

- un champ formulaire
- ajouté automatiquement par **Symfony**
- utilisé pour assurer la sécurité lors de l'envoi du formulaire

Symfony

Contenu de la vue `add.html.twig`

```
{% extends 'base.html.twig' %}

{% block title %}Hello PersonneController!
{% endblock %}

{% block body %}

    <h1>Hello
        {{ controller_name }}!
    </h1>

    {{ form(form) }}

{% endblock %}
```

Symfony

Trois problématiques

- Le formulaire ne permet pas d'ajouter les valeurs saisies dans la base de données
- Le formulaire est mal présenté (ce n'est pas le développeur qui organise les éléments du formulaire)
- La construction du formulaire doit être faite dans une autre classe (pas dans le contrôleur)

Modifions l'action pour permettre l'ajout dans la base de données

```
public function addForm(EntityManagerInterface $entityManager, Request
   $request)
{
    $personne = new Personne();
    $form = $this->createFormBuilder($personne)
        ->add('nom', TextType::class, array('required' => true))
        ->add('prenom', TextType::class)
        ->add('save', SubmitType::class, ['label' => 'Ajouter une
           personne'])
        ->getForm();
     // pour récupérer les variables de la requête POST
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $personne = $form->getData();
        $entityManager->persist($personne);
        $entityManager->flush();
        return $this->redirectToRoute('personne_show_all');
    }
    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Dans `add.html.twig`**, remplaçons** {{ `form(form)` }} **par le contenu suivant**

```
<div>
        {{ form_start(form) }}
        {{ form_errors(form) }}
        <div>
                {{ form_label(form.nom, "Nom de la personne") }}
                {{ form_errors(form.nom) }}
                <div>
                        {{ form_widget(form.nom) }}
                </div>
        </div>
        <div>
                {{ form_label(form.prenom, "Prénom de la personne") }}
                {{ form_errors(form.prenom) }}
                <div>
                        {{ form_widget(form.prenom) }}
                </div>
        </div>
        <div>
                {{ form_rest(form) }}
        </div>
        {{ form_end(form) }}
</div>
```

Symfony

`form_start()`

- L'équivalent de `<form>` en HTML.
- Acceptant deux paramètres.
  - Le premier est la variable formulaire (passé depuis le contrôleur).
  - Le deuxième contient l'index `attr` des paramètres (nom de la classe...).

Symfony

`form_start()`

- L'équivalent de `<form>` en HTML.
- Acceptant deux paramètres.
  - Le premier est la variable formulaire (passé depuis le contrôleur).
  - Le deuxième contient l'index `attr` des paramètres (nom de la classe...).

Exemple

```
{{ form_start(form, { 'attr': { 'class': 'formPersonne' } }) }}
```

Symfony

`form_errors()`

Elle affiche les erreurs relatives au champ donné en argument.

Symfony

`form_errors()`

Elle affiche les erreurs relatives au champ donné en argument.

`form_label()`

- affiche le label HTML d'un élément du formulaire.
- accepte deux paramètres :
  - Le premier est le champ concerné par le label.
  - Le deuxième est le contenu du label.

Symfony

`form_errors()`

Elle affiche les erreurs relatives au champ donné en argument.

`form_label()`

- affiche le label HTML d'un élément du formulaire.
- accepte deux paramètres :
  - Le premier est le champ concerné par le label.
  - Le deuxième est le contenu du label.

Exemple

```
{{ form_label(form.nom, "Nom de la personne") }}
```

Symfony

`form_widget()`

Elle affiche le champ du formulaire lui-même (soit `<input>`, soit `<select>`...).

Symfony

`form_widget()`

Elle affiche le champ du formulaire lui-même (soit `<input>`, soit `<select>`...).

`form_row()`

= `form_label` + `form_errors` + `form_widget`

Symfony

`form_widget()`

Elle affiche le champ du formulaire lui-même (soit `<input>`, soit `<select>`...).

`form_row()`

= `form_label` + `form_errors` + `form_widget`

`form_rest()`

Affiche le reste du formulaire (les champs que nous n'avons pas précisé).

Symfony

`form_widget()`

Elle affiche le champ du formulaire lui-même (soit `<input>`, soit `<select>`...).

`form_row()`

= `form_label` + `form_errors` + `form_widget`

`form_rest()`

Affiche le reste du formulaire (les champs que nous n'avons pas précisé).

`form_end()`

L'équivalent de `</form>` en HTML

Symfony

Pour générer une classe formulaire

- exécutez `php bin/console make:form`
- répondez à `The name of the form class` par `Personne`
- et à `The name of Entity or fully qualified model class name that the new form will be bound to (empty for none)` par `Personne`

Symfony

Pour générer une classe formulaire

- exécutez `php bin/console make:form`
- répondez à `The name of the form class` par `Personne`
- et à `The name of Entity or fully qualified model class name that the new form will be bound to (empty for none)` par `Personne`

Constat

Une classe `PersonneType` a été créée dans `src/Form`

Contenu de `PersonneType.php`

```php
namespace App\Form;

use App\Entity\Personne;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class PersonneType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
       $options)
    {
        $builder
            ->add('nom')
            ->add('prenom')
        ;
    }
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Personne::class,
        ]);
    }
}
```

Symfony

Quelques valeurs possibles pour la clé du troisième paramètre de la méthode `add`

- `required`
- `label`
- `constraints`
- `choices`
- `class`
- `multiple`
- ...

Symfony

Remarques

- Dans les méthodes `add`, les types ne sont pas précisés.
- **Symfony** s'appuie sur les annotations (définies dans les entités) pour la génération des types.

Symfony

Ajoutons le bouton et précisons les types dans `PersonneType.php`

```
public function buildForm(FormBuilderInterface $builder, array
   $options)
{
    $builder
        ->add('nom', TextType::class, array('required' => true)
           )
        ->add('prenom', TextType::class)
        ->add('save', SubmitType::class, ['label' => 'Ajouter
           une personne']);
}
```

Symfony

Ajoutons le bouton et précisons les types dans `PersonneType.php`

```
public function buildForm(FormBuilderInterface $builder, array
   $options)
{
    $builder
        ->add('nom', TextType::class, array('required' => true)
           )
        ->add('prenom', TextType::class)
        ->add('save', SubmitType::class, ['label' => 'Ajouter
           une personne']);
}
```

Les `use` **nécessaires**

```
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
```

Symfony

Remarque

Pensez à supprimer les attributs statiques et les attributs qui ne doivent pas être renseignés par l'utilisateur

Utilisons la classe formulaire dans `PersonneController`

```
public function addForm(EntityManagerInterface $entityManager, Request
   $request)
{
    $personne = new Personne();
    $form = $this->get('form.factory')->create(PersonneType::class,
       $personne);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $personne = $form->getData();
        $entityManager->persist($personne);
        $entityManager->flush();
        return $this->redirectToRoute('personne_show_all');
    }
    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Utilisons la classe formulaire dans `PersonneController`

```
public function addForm(EntityManagerInterface $entityManager, Request
   $request)
{
    $personne = new Personne();
    $form = $this->get('form.factory')->create(PersonneType::class,
       $personne);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $personne = $form->getData();
        $entityManager->persist($personne);
        $entityManager->flush();
        return $this->redirectToRoute('personne_show_all');
    }
    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Le `use` nécessaire

```
use App\Form\PersonneType;
```

Ou le raccourci

```
public function addForm(EntityManagerInterface $entityManager, Request
   $request)
{
    $personne = new Personne();
    $form = $this->createForm(PersonneType::class, $personne);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $personne = $form->getData();
        $entityManager->persist($personne);
        $entityManager->flush();
        return $this->redirectToRoute('personne_show_all');
    }
    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Ou le raccourci

```
public function addForm(EntityManagerInterface $entityManager, Request
   $request)
{
    $personne = new Personne();
    $form = $this->createForm(PersonneType::class, $personne);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $personne = $form->getData();
        $entityManager->persist($personne);
        $entityManager->flush();
        return $this->redirectToRoute('personne_show_all');
    }
    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Le `use` nécessaire

```
use App\Form\PersonneType;
```

Symfony

Il est possible de

- séparer la préparation et le traitement de formulaire en deux méthodes
- modifier la méthode d'un formulaire
- modifier l'action

Symfony

Modifions `addForm()` pour qu'elle affiche seulement le formulaire

```
/**
 * @Route("/personne/add", name="personne_add", methods={"GET
    "})
 */
public function addForm()
{
    $personne = new Personne();
    $form = $this->createForm(PersonneType::class, $personne, [
        'action' => $this->generateUrl('personne_add_post'),
        'method' => 'POST',
    ]);
    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Symfony

Ajoutons `addPersonne()` pour récupérer les données du formulaire

```
/**
 * @Route("/personne/add", name="personne_add_post", methods={"POST"})
 */
function addPersonne(EntityManagerInterface $entityManager,  Request
   $request)
{
    $form = $this->createForm(PersonneType::class);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $personne = $form->getData();
        $entityManager->persist($personne);
        $entityManager->flush();
        return $this->redirectToRoute('personne_show_all');
    }
    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

On peut aussi indiquer la méthode et l'action dans la vue

```
{{ form_start(form, {'action': path('personne_add_post'), '
   method': 'POST'}) }}
```

On peut aussi indiquer la méthode et l'action dans la vue

```
{{ form_start(form, {'action': path('personne_add_post'), '
   method': 'POST'}) }}
```

Contenu de l'action

```
/**
 * @Route("/personne/add", name="personne_add", methods={"GET
    "})
 */
public function addForm()
{
    $personne = new Personne();
    $form = $this->createForm(PersonneType::class, $personne);
    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Symfony

Pour la suite, nous utiliserons la version avec une seule méthode (pour l'ajout)

```
/**
 * @Route("/personne/add", name="personne_add")
 */
function addForm(EntityManagerInterface $entityManager, Request
   $request)
{
    $personne = new Personne();
    $form = $this->createForm(PersonneType::class, $personne);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $personne = $form->getData();
        $entityManager->persist($personne);
        $entityManager->flush();
        return $this->redirectToRoute('personne_show_all');
    }
    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Symfony

Pourquoi ?

- Une entité peut être en relation avec une autre entité.
- Par exemple, une entité `Personne` peut être en relation avec l'entité `Adresse`.
- En créant un objet de type `Personne` via le formulaire, on doit permettre à l'utilisateur de lui associer un objet de l'entité `adresse`

Symfony

Comment ?

- Créez l'entité `Adresse`
- Établissez la relation `OneToOne` avec l'entité `Personne`
- Mettez à jour la base de données
- Créez la classe formulaire `AdresseType`
- Intégrez `AdresseType` dans `PersonneType`

Symfony

Pour créer l'entité `Adresse`

- Exécutez la commande `php bin/console make:entity`
- Cette entité a trois attributs :
  - `rue` (`string` de taille 30),
  - `codePostal` (`string` de taille 5) et
  - `ville` (`string` de taille 30)

Symfony

Pour ajouter `Adresse` dans `Personne`

- exécutez la commande `php bin/console make:entity`
- répondez à `Class name of the entity to create or update` par `Personne`
- répondez à `New property name` par `adresse`
- répondez à `Field type` par `OneToOne`
- répondez à `What class should this entity be related to?` par `Adresse`
- répondez à `Is the Personne.adresse property allowed to be null (nullable)?` par `yes`
- répondez à `Do you want to add a new property to Adresse so that you can access/update Personne` par `no`
- cliquez sur `entrée` pour répondre à `Add another property?`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Vérifier les modifications avec la console `MySQL` ou `phpMyAdmin`

Symfony

Pour générer une classe formulaire `AdresseType`

- exécutez `php bin/console make:form`
- répondez à `The name of the form class` par `Adresse`
- et à `The name of Entity or fully qualified model class name that the new form will be bound to (empty for none)` par `Adresse`

Symfony

Pour générer une classe formulaire `AdresseType`

- exécutez `php bin/console make:form`
- répondez à `The name of the form class` par `Adresse`
- et à `The name of Entity or fully qualified model class name that the new form will be bound to (empty for none)` par `Adresse`

Constat

Une classe `AdresseType` a été créée dans `src/Form`

Symfony

Contenu de `AdresseType.php`

```
class AdresseType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
       $options)
    {
        $builder
            ->add('rue')
            ->add('codePostal')
            ->add('ville')
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Adresse::class,
        ]);
    }
}
```

Modifions `AdresseType.php`

```
class AdresseType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
       $options)
    {
        $builder
            ->add('rue', TextType::class)
            ->add('codePostal', TextType::class)
            ->add('ville', TextType::class)
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Adresse::class,
        ]);
    }
}
```

Modifions `AdresseType.php`

```
class AdresseType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
       $options)
    {
        $builder
            ->add('rue', TextType::class)
            ->add('codePostal', TextType::class)
            ->add('ville', TextType::class)
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Adresse::class,
        ]);
    }
}
```

Le `use` nécessaire

```
use Symfony\Component\Form\Extension\Core\Type\TextType;;
```

Symfony

Intégrons `Adresse` dans `PersonneType.php`

```
public function buildForm(FormBuilderInterface
   $builder, array $options)
{
    $builder
        ->add('nom', TextType::class, array('
           required' => true))
        ->add('prenom', TextType::class)
        ->add('adresse', AdresseType::class)
        ->add('save', SubmitType::class, ['label' =>
            'Ajouter une personne']);
}
```

Symfony

Comment ?

- Créez l'entité `Sport`
- Établissez la relation `ManyToMany` avec l'entité `Personne`
- Mettez à jour la base de données
- Créez la classe formulaire `SportType`
- Intégrez `SportType` dans `PersonneType`

Symfony

Pour créer l'entité `Sport`

- Exécutez la commande `php bin/console make:entity`
- Cette entité a un seul attribut :
  - `name` de type `string`

Symfony

Pour ajouter `Sport` dans `Personne`

- exécutez la commande `php bin/console make:entity`
- répondez à `Class name of the entity to create or update` par `Sport`
- répondez à `New property name` par `sports`
- répondez à `Field type` par `ManyToMany`
- répondez à `What class should this entity be related to?` par `Sport`
- répondez à `Is the Personne.sports property allowed to be null (nullable)?` par `yes`
- répondez à `Do you want to add a new property to Sport so that you can access/update Personne` par `no`
- cliquez sur `entrée` pour répondre à `Add another property?`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Symfony

Pour régénérer les tables dans la base de données, exécutez

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Vérifier les modifications avec la console `MySQL` ou `phpMyAdmin`

Symfony

Pour générer une classe formulaire `SportType`

- exécutez `php bin/console make:form`
- répondez à `The name of the form class` par `Sport`
- et à `The name of Entity or fully qualified model class name that the new form will be bound to (empty for none)` par `Sport`

Symfony

Pour générer une classe formulaire `SportType`

- exécutez `php bin/console make:form`
- répondez à `The name of the form class` par `Sport`
- et à `The name of Entity or fully qualified model class name that the new form will be bound to (empty for none)` par `Sport`

Constat

Une classe `SportType` a été créée dans `src/Form`

Symfony

Contenu de `SportType.php`

```
class SportType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
       $options)
    {
        $builder
            ->add('name')
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Sport::class,
        ]);
    }
}
```

Modifions `SportType.php`

```
class SportType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
       $options)
    {
        $builder
            ->add('name', TextType::class)
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Sport::class,
        ]);
    }
}
```

Modifions `SportType.php`

```
class SportType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
       $options)
    {
        $builder
            ->add('name', TextType::class)
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Sport::class,
        ]);
    }
}
```

Le `use` nécessaire

```
use Symfony\Component\Form\Extension\Core\Type\TextType;;
```

Symfony

Intégrons `Sport` **dans** `PersonneType.php`

```
function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('nom', TextType::class, array('required' => true))
        ->add('prenom', TextType::class)
        ->add('adresse', AdresseType::class)
        ->add('sports', CollectionType::class, [
            'entry_type' => SportType::class,
            'entry_options' => ['label' => false],
            'allow_add' => true,
            'allow_delete' => true,
        ])
        ->add('save', SubmitType::class, ['label' => 'Ajouter une
          personne']);
}
```

Symfony

Intégrons `Sport` dans `PersonneType.php`

```
function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('nom', TextType::class, array('required' => true))
        ->add('prenom', TextType::class)
        ->add('adresse', AdresseType::class)
        ->add('sports', CollectionType::class, [
            'entry_type' => SportType::class,
            'entry_options' => ['label' => false],
            'allow_add' => true,
            'allow_delete' => true,
        ])
        ->add('save', SubmitType::class, ['label' => 'Ajouter une
          personne']);
}
```

Le `use` nécessaire

```
use Symfony\Component\Form\Extension\Core\Type\CollectionType;
```

Symfony

Explication

- `allow_add` : permet d'autoriser l'ajout d'un nouveau sport
- `allow_delete` : indique que si un élément d'une collection n'est pas envoyé lors de la soumission, les données associées sont supprimées de la collection sur le serveur.
- `entry_type` : précise le nom du formulaire imbriqué
- `entry_options` : permet d'ajouter certaines options (de ne pas afficher le label par exemple)

Symfony

Malheureusement, ceci ne suffit pas pour afficher le formulaire relatif au Sport.

Symfony

Malheureusement, ceci ne suffit pas pour afficher le formulaire relatif au Sport.

Pas de message d'erreur, mais pas de formulaire affiché, seulement le label sport

Symfony

Malheureusement, ceci ne suffit pas pour afficher le formulaire relatif au Sport.

Pas de message d'erreur, mais pas de formulaire affiché, seulement le label sport

En effet, **Symfony** ignore le nombre de formulaire sport à afficher pour chaque `Personne`.

Symfony

Malheureusement, ceci ne suffit pas pour afficher le formulaire relatif au Sport.

Pas de message d'erreur, mais pas de formulaire affiché, seulement le label sport

En effet, **Symfony** ignore le nombre de formulaire sport à afficher pour chaque `Personne`.

Il nous faut un script (**JavaScript** ou **jQuery**) pour permettre à l'utilisateur d'ajouter un nombre variable de sports.

Symfony

Ajoutons le contenu suivant à notre formulaire

```
<ul class="sports" data-prototype="{{ form_widget(form.sports.
   vars.prototype)|e('html_attr') }}">
</ul>
```

Symfony

Ajoutons le contenu suivant à notre formulaire

```
<ul class="sports" data-prototype="{{ form_widget(form.sports.
   vars.prototype)|e('html_attr') }}">
</ul>
```

Après le block `body` **de** `add.html.twig`**, ajoutons la référence vers le fichier** `script.js`

```
{% block javascripts %}
        <script src="//ajax.googleapis.com/ajax/libs/jquery
           /1.11.1/jquery.min.js">
        </script>
        <script src="{{ asset('js/script.js') }}">
        </script>
{% endblock %}
```

Symfony

Contenu de `script.js` défini dans `public/js` (première partie)

```
var $collectionHolder;
var $boutonAjouter = $('<button type="button" class="add_sport_link">
   Ajouter sport</button>');
var $nouveau = $('<li></li>').append($boutonAjouter);

jQuery(document).ready(function () {
    $collectionHolder = $('ul.sports');
    $collectionHolder.find('li').each(function () {
        addSportFormDeleteLink($(this));
    });
    $collectionHolder.append($nouveau);
    $collectionHolder.data('index', $collectionHolder.find('input').
       length);
    $boutonAjouter.on('click', function (e) {
        addSportForm($collectionHolder, $nouveau);
    });
```

Symfony

Contenu de `script.js` défini dans `public/js` (deuxième partie)

```
    function addSportForm($collectionHolder, $newLinkLi) {
        var prototype = $collectionHolder.data('prototype');
        var index = $collectionHolder.data('index');
        var newForm = prototype;
        newForm = newForm.replace(/__name__/g, index);
        $collectionHolder.data('index', index + 1);
        var $newFormLi = $('<li></li>').append(newForm);
        $newLinkLi.before($newFormLi);
        addSportFormDeleteLink($newFormLi);
    }

    function addSportFormDeleteLink($tagFormLi) {
        var $removeFormButton = $('<button type="button">Supprimer
           sport</button>');
        $tagFormLi.append($removeFormButton);

        $removeFormButton.on('click', function (e) {
            $tagFormLi.remove();
        });
    }
});
```

Pour seulement sélectionner des sports déjà existants (sans créer de nouveaux), on modifie la classe formulaire

```
public function buildForm(FormBuilderInterface $builder, array $options
    )
{
    $builder
        ->add('nom', TextType::class, array('required' => true))
        ->add('prenom', TextType::class)
        ->add('adresse', AdresseType::class)
        ->add('sports', EntityType::class, [
            'class' => Sport::class,
            'choice_label' => 'name',
            'query_builder' => function (EntityRepository $repo) {
                return $repo->createQueryBuilder('s');
            },
            'label'     => 'Sports préférés',
            'multiple' => true
        ])
        ->add('save', SubmitType::class, ['label' => 'Ajouter une
          personne']);
}
```

Pour seulement sélectionner des sports déjà existants (sans créer de nouveaux), on modifie la classe formulaire

```
public function buildForm(FormBuilderInterface $builder, array $options
    )
{
    $builder
        ->add('nom', TextType::class, array('required' => true))
        ->add('prenom', TextType::class)
        ->add('adresse', AdresseType::class)
        ->add('sports', EntityType::class, [
            'class' => Sport::class,
            'choice_label' => 'name',
            'query_builder' => function (EntityRepository $repo) {
                return $repo->createQueryBuilder('s');
            },
            'label'     => 'Sports préférés',
            'multiple' => true
        ])
        ->add('save', SubmitType::class, ['label' => 'Ajouter une
          personne']);
}
```

Les `use` nécessaires

```
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
use Doctrine\ORM\EntityRepository;
```

Symfony

Les options de `EntityType`

- `class` : précise l'entité à sélectionner
- `choice_label` : indique l'attribut de l'entité `Sport` à afficher dans le `select` du formulaire
- `multiple` : permet la sélection multiple
- `query_builder` : précise la requête qui récupère la liste des sports
- `label` : indique le libellé de la liste déroulante

Pour afficher une collection (statique) de sports

```
public function buildForm(FormBuilderInterface $builder, array $options
    )
{
    $builder
        ->add('nom', TextType::class, array('required' => true))
        ->add('prenom', TextType::class)
        ->add('adresse', AdresseType::class)
        ->add('sports', ChoiceType::class, [
            'choices' => [
                'Foot' => 'foot',
                'Tennis' => 'Tennis',
                'Other' => null,
            ],
        ])
        ->add('save', SubmitType::class, ['label' => 'Ajouter une
          personne']);
}
```

Pour afficher une collection (statique) de sports

```
public function buildForm(FormBuilderInterface $builder, array $options
    )
{
    $builder
        ->add('nom', TextType::class, array('required' => true))
        ->add('prenom', TextType::class)
        ->add('adresse', AdresseType::class)
        ->add('sports', ChoiceType::class, [
            'choices' => [
                'Foot' => 'foot',
                'Tennis' => 'Tennis',
                'Other' => null,
            ],
        ])
        ->add('save', SubmitType::class, ['label' => 'Ajouter une
          personne']);
}
```

Le `use` nécessaire

```
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
```

Symfony

Le concept d'héritage

- syntaxiquement différent du concept d'héritage connu dans le modèle objet
- sémantiquement identique, permettant
  - la réutilisation du code
  - à un élément héritant d'avoir ses propres particularités

Symfony

Exemple

- Si on voulait avoir un formulaire pour créer des nouvelles personnes sans (leur affecter une adresse ou un sport)
- Une première solution consiste à définir un nouveau formulaire sans les sous-formulaires pour adresse et sport
- Uns solution plus consistante consiste à construire un nouveau formulaire à partir du premier tout (en utilisant l'héritage) en supprimant les sous-formulaires relatifs à mes objets.

Symfony

Contenu de la classe formulaire : `OnlyPersonneType.php` (non associée à une entité)

```php
namespace App\Form;

use Symfony\Component\Form\FormBuilderInterface;

class OnlyPersonneType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
       $options)
    {
        $builder
            ->remove('adresse')
            ->remove('sports');
    }
    public function getParent()
    {
        return PersonneType::class;
    }
}
```

Symfony

Comment on a défini l'héritage ?

- En effet , la méthode `getParent()` définit l'association d'héritage entre `PersonneType` et `OnlyPersonneType`
- Le compilateur commencera par exécuter la méthode `buildForm` du formulaire parent (`PersonneType`) ensuite exécute celle de `OnlyPersonneType` qui supprimera les sous-formulaires relatifs aux `Adresse` et `Sport`.

Symfony

Comment on a défini l'héritage ?

- En effet , la méthode `getParent()` définit l'association d'héritage entre `PersonneType` et `OnlyPersonneType`
- Le compilateur commencera par exécuter la méthode `buildForm` du formulaire parent (`PersonneType`) ensuite exécute celle de `OnlyPersonneType` qui supprimera les sous-formulaires relatifs aux `Adresse` et `Sport`.

Pour tester

Remplacez `PersonneType` par `OnlyPersonneType` dans `PersonneController`.

Symfony

Principe

- On peut déclencher des événements avant ou après un changement survenu dans un formulaire ou son objet associé
- On peut aussi dynamiser l'affichage de notre formulaire

Symfony

Exemple

- Si on voulait permettre la modification d'adresse seulement aux personnes qui en ont déjà une.
- Les personnes qui n'ont pas d'adresse ne peuvent pas en créer une.

Commençons par modifier la classe formulaire `PersonneType.php`

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('nom', TextType::class, array('required' => true))
        ->add('prenom', TextType::class)
        ->addEventListener(FormEvents::PRE_SET_DATA, function (FormEvent $event) {
            $personne = $event->getData();
            $form = $event->getForm();
            if ($personne != null && ($personne->getId() == null || $personne->getAdresse() !=
               null)) {
                $form->add('adresse', AdresseType::class);
            }
        })
        ->add('sports', EntityType::class, [
            'class' => Sport::class,
            'choice_label' => 'name',
            'query_builder' => function (EntityRepository $repo) {
                return $repo->createQueryBuilder('s');
            },
            'label'     => 'Sports préférés',
            'multiple' => true
        ])
        ->add('save', SubmitType::class, ['label' => 'Ajouter une personne']);
}
```

Commençons par modifier la classe formulaire `PersonneType.php`

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('nom', TextType::class, array('required' => true))
        ->add('prenom', TextType::class)
        ->addEventListener(FormEvents::PRE_SET_DATA, function (FormEvent $event) {
            $personne = $event->getData();
            $form = $event->getForm();
            if ($personne != null && ($personne->getId() == null || $personne->getAdresse() !=
               null)) {
                $form->add('adresse', AdresseType::class);
            }
        })
        ->add('sports', EntityType::class, [
            'class' => Sport::class,
            'choice_label' => 'name',
            'query_builder' => function (EntityRepository $repo) {
                return $repo->createQueryBuilder('s');
            },
            'label'     => 'Sports préférés',
            'multiple' => true
        ])
        ->add('save', SubmitType::class, ['label' => 'Ajouter une personne']);
}
```

Les `use` **nécessaires**

```
use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;
```

Symfony

Les différents évènements

- `POST_SET_DATA`
- `PRE_SUBMIT`
- `SUBMIT`
- `POST_SUBMIT`

Symfony

Dans le contrôleur, modifions la méthode `updatePersonne`

```
/**
 * @Route("/personne/edit/{id}", name="personne_update")
 */
public function updatePersonne(Personne $personne,
   EntityManagerInterface $entityManager, Request $request)
{
    $form = $this->createForm(PersonneType::class, $personne);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $personne = $form->getData();
        $entityManager->flush();
        return $this->redirectToRoute('personne_show_all');
    }

    return $this->render('personne/add.html.twig', [
        'controller_name' => 'PersonneController',
        'form' => $form->createView(),
    ]);
}
```

Symfony

Champs non-mappé ?

élément de formulaire qui n'a pas de correspondance dans l'entité **Doctrine**.

Symfony

Champs non-mappé ?

élément de formulaire qui n'a pas de correspondance dans l'entité **Doctrine**.

Exemple

Ajoutons un **checkbox** à cocher par l'utilisateur pour attester l'exactitude de ces informations saisies.

Symfony

Dans la classe formulaire, ajoutons l'élément `accepter` dans la méthode `buildForm`

```
->add('accepter', CheckboxType::class, ['mapped' => false])
->addEventListener(FormEvents::PRE_SUBMIT, function (FormEvent $event)
   {
     $personne = $event->getData();
     if (!isset($personne['accepter']) || !$personne['accepter']) {
         exit;
     }
})
```

Symfony

Dans la classe formulaire, ajoutons l'élément `accepter` dans la méthode `buildForm`

```
->add('accepter', CheckboxType::class, ['mapped' => false])
->addEventListener(FormEvents::PRE_SUBMIT, function (FormEvent $event)
   {
     $personne = $event->getData();
     if (!isset($personne['accepter']) || !$personne['accepter']) {
         exit;
     }
})
```


Dans la vue, on personnalise l'affichage de notre nouvel élément

```
<div>
        {{ form_label(form.accepter, "je déclare sur l'honneur que les
           informations ci-dessus sont exactes") }}
        {{ form_widget(form.accepter) }}
</div>
```

Symfony

Deux règles d'or de **Microsoft**

- never trust user input
- and always check data as it moves from an untrusted to a trusted domain

Symfony

Principe

- La validation des objets (entité ou autre) se réalise avec le composant `Validator` de Symfony
- Pour décider si un objet (ou un formulaire) est valide, il faut définir des règles et et les rattacher aux objets
- Par exemple
  - un mot de passe doit contenir au moins 8 caractères
  - une note est comprise entre 0 et 20
  - ...
- La validation des formulaires permet de garder la base de données dans un état cohérent.

Symfony

Comment ?

- En utilisant les annotations
- On peut aussi les externaliser dans un fichier
  - `XML`
  - `YML`
  - `PHP`

Symfony

Pour Doctrine

On a utilisé le namespace `use Doctrine\ORM\Mapping as ORM;`

Symfony

Pour Doctrine

On a utilisé le namespace `use Doctrine\ORM\Mapping as ORM;`

Pour le validator

```
use Symfony\Component\Validator\Constraints as
Assert;
```

Symfony

Syntaxe

- La forme réduite : `@Assert\Contrainte (valeur de l'option par défaut)`
- La forme étendue : `@Assert\Contrainte (option1="valeur1", ... optionN="valeurN")`

Symfony

Syntaxe

- La forme réduite : `@Assert\Contrainte (valeur de l'option par défaut)`
- La forme étendue : `@Assert\Contrainte (option1="valeur1", ... optionN="valeurN")`

Exemple

- La forme réduite : `@Assert\Choice("homme", "femme")`
- La forme étendue : `@Assert\Length(min=10,minMessage= "Votre mot de passe {{ value }} ne contient pas {{ limit }} caractères."))`

Symfony

Remarques

- Les contraintes varient selon le type du champ
- Chaque contrainte a plusieurs options dont une par défaut
- Un champ peut être annotés par plusieurs contraintes

Symfony

Les contraintes générales

- `Blank` : vérifie si la valeur d'un champ est une chaîne vide ou null (`NotBlank` est l'inverse)
- `IsTrue` : vérifie si la valeur d'un champ est true, 1 ou '1' (pareillement pour `IsFalse`)
- `Type(nomType)` : vérifie si la valeur d'un champ est de type `nomType`
- `NotNull` : vérifie si la valeur d'un champ est différente de `null` (pareillement pour `IsNull`)

Symfony

Les contraintes sur les chaînes de caractères

- `Length` : vérifie si la valeur d'un champ vérifie certaines conditions. Elle accepte plusieurs options :
  - `min` et `minMessage` (le message d'erreur à afficher si la contrainte min n'est pas respectée)
  - `max` et `maxMessage`
  - ...
- `Url` : vérifie si la valeur est une adresse URL valide
- `Ip` : vérifie si la valeur d'un champ est une adresse IP
- ...

Symfony

Les contraintes sur les nombres

- `Range` : comme `Length` mais elle vérifie la valeur et pas la longueur. Elle peut aussi prendre les mêmes paramètres que `Length`
- `DivisibleBy`
- `GreaterThan`
- `EqualTo`
- `Positive`
- `PositiveOrZero`
- ...

Symfony

Les contraintes sur les dates

- `Date` : vérifie si la valeur est un objet de type `Datetime`, ou une chaîne de caractères du type `YYYY-MM-DD`
- `Time` : vérifie si la valeur est un objet de type `Datetime`, ou une chaîne de caractères du type `HH:MM:SS`
- `DateTime` : vérifie si la valeur est un objet de type `Datetime`, ou une chaîne de caractères du type `YYYY-MM-DD HH:MM:SS`

Symfony

Autres contraintes

- `Currency` : vérifie si la valeur respecte le codage `3-letter ISO 4217` (EUR, USD...)
- `Isbn`
- `Country` : vérifie si la valeur respecte le codage `ISO 3166-1 alpha-2` (FR, US, TN, ES...)
- ...

Symfony

La liste complète

`https://symfony.com/doc/current/validation.html`

Symfony

Exemple ; ajoutons les contraintes suivantes sur le nom dans l'entité `Personne`

```
/**
     * @ORM\Column(type="string", length=255)
     * @Assert\Length(
     *      min = 2,
     *      max = 20,
     *      minMessage = "Votre nom doit contenir au moins {{
        limit }} caractères",
     *      maxMessage = "Votre nom doit contenir au plus {{
        limit }} caractères",
     *      allowEmptyString = false
     * )
     * @Assert\Type(
     *     type={"alpha", "digit"},
     *     message="Votre nom {{ value }} doit contenir
        seulement {{ type }}."
     * )
 */
private $nom;
```

Symfony

Pour vérifier la validité du formulaire

```
if ($form->isSubmitted() && $form->isValid()) {
    ...
}
```

Symfony

Pour vérifier la validité du formulaire

```
if ($form->isSubmitted() && $form->isValid()) {
    ...
}
```

Ou

```
$errors = $validator->validate($personne);
if (count($errors) > 0) {
    ...
}
```

Les contraintes peuvent être définies dans la classe formulaire

```
->add('nom', TextType::class, [
    'required' => true,
    'constraints' => [
        new Length([
            'min' => 3,
            'max' => 20,
            'minMessage' => "Votre nom doit contenir au moins {{ limit
              }} caractères",
            'maxMessage' => "Votre nom doit contenir au plus {{ limit
              }} caractères",
        ]),
        new Type([
            'type' => '{"alpha", "digit"}',
            'message' => "Votre nom {{ value }} doit contenir seulement
               {{ type }}."
        ])
    ]
])
```

Les contraintes peuvent être définies dans la classe formulaire

```
->add('nom', TextType::class, [
    'required' => true,
    'constraints' => [
        new Length([
            'min' => 3,
            'max' => 20,
            'minMessage' => "Votre nom doit contenir au moins {{ limit
              }} caractères",
            'maxMessage' => "Votre nom doit contenir au plus {{ limit
              }} caractères",
        ]),
        new Type([
            'type' => '{"alpha", "digit"}',
            'message' => "Votre nom {{ value }} doit contenir seulement
               {{ type }}."
        ])
    ]
])
```

Les `use` nécessaires

```
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\Type;
```

Symfony

On peut aussi annoter une entité

```
/**
 * @ORM\Entity
 * @UniqueEntity(fields="name", message="Ce sport existe
    deja")
 */
class Sport
{
    ...
}
```

Symfony

On peut aussi annoter une entité

```
/**
 * @ORM\Entity
 * @UniqueEntity(fields="name", message="Ce sport existe
    deja")
 */
class Sport
{
    ...
}
```

L'annotation `@UniqueEntity` permet de préciser qu'un champ est unique.

Symfony

Dans ce cas, on ne peut insérer deux personnes avec le même couple (`nom`, `prénom`)

```
/**
 * @ORM\Entity
 * @UniqueEntity(fields={"nom", "prenom"}, message="
    impossible")
 */
class Personne
{
    ...
}
```

Symfony

On peut aussi valider selon une contrainte personnalisée

- Créer une contrainte
- Créer un validateur
- Utiliser nos contraintes comme des annotations

Symfony

Exemple

- On va supposer que les nom et prénom d'un utilisateur ne doivent pas contenir de chiffres
- On va donc définir une contrainte `OnlyCharacterAndSpace`
- Et on va utiliser cette contrainte comme annotation (`@OnlyCharacterAndSpace()`) sur les deux attributs `nom` et `prénom` avec cette contrainte

Symfony

Créons le fichier `OnlyCharacterAndSpace.php` **dans** `src/Validator`

```php
<?php

namespace App\Validator\Constraints;

use Symfony\Component\Validator\Constraint;

/**
 * @Annotation
 */
class OnlyCharacterAndSpace extends Constraint
{
    public $message = "La chaine '{{ string }}' doit
       contenir seulement des lettres et des espaces";
}
```

Symfony

Créons aussi le validateur `OnlyCharacterAndSpaceValidator.php`

```php
<?php

namespace App\Validator\Constraints;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class OnlyCharacterAndSpaceValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        if (!preg_match('/^[a-zA-Z ]+$/', $value)) {
            $this->context->buildViolation($constraint->message)
                ->setParameter('{{ string }}', $value)
                ->addViolation();
        }
    }
}
```

Symfony

Attention

- Par convention, le nom du validateur doit être ainsi : `NomContrainteValidator`
- Le validateur qui hérite de `ConstraintValidator` doit redéfinir la méthode `validate`
- Le paramètre `$value` correspondra à la valeur saisie par l'utilisateur dans la zone annotée par notre contrainte

Symfony

Utilisons cette nouvelle contrainte dans `Personne.php`

```
/**
 * @ORM\Column(type="string", length=255)
 * @MyAssert\OnlyCharacterAndSpace
 */
private $nom;

/**
 * @ORM\Column(type="string", length=255)
 * @MyAssert\OnlyCharacterAndSpace
 */
private $prenom;
```

Symfony

Utilisons cette nouvelle contrainte dans `Personne.php`

```
    /**
     * @ORM\Column(type="string", length=255)
     * @MyAssert\OnlyCharacterAndSpace
     */
    private $nom;

    /**
     * @ORM\Column(type="string", length=255)
     * @MyAssert\OnlyCharacterAndSpace
     */
    private $prenom;
```

Le `use` **nécessaire**

```
use App\Validator\Constraints as MyAssert;
```

Symfony

Commande **Symfony** pour générer le CRUD selon une entité **Doctrine** (`Adresse`)

```
php bin/console make:crud
```

Symfony

Commande **Symfony** pour générer le CRUD selon une entité **Doctrine** (`Adresse`)

```
php bin/console make:crud
```

Résultat

- Contrôleur `AdresseController` généré
- Formulaire `AdresseType` généré
- Plusieurs vues